SimpleFleet	**SimpleFleet**
	D4.2: TrafficApps

Author(s)	Athanasios Mantes (TALENT)			
Project	SimpleFleet – Democratizing Fleet Management			
Date	*Contractual:*	30.10.2013	*Actual:*	28.10.2013
Project Coordinator	Rüdiger Ebendt Deutsches Zentrum für Luft- und Raumfahrt (DLR) Tel: +49 30 67055 287 E-mail: ruediger.ebendt@dlr.de			

Abstract	This deliverable describes the SimpleFleet TrafficApps, a collection of software modules, using data from the SimpleFleet TrafficStore and functionality of the SimpleFleet TrafficAPI/TrafficSDK to provide specific traffic functionality. These Apps can be combined and extended with minimal effort, to vehicle fleet management programs providing full software functionality.
Keyword list	TrafficAPI, TrafficSDK, TrafficStore, Routing, Isochrones, FCD
Nature of deliverable	Report
Dissemination	Confidential

Project financially supported by	
SEVENTH FRAMEWORK PROGRAMME	**European Commission** **DG CONNECT**
Project number 296423 **FP7-ICT-2011-SME-DCL**	

Control sheet

Version history			
Version number	**Date**	**Main author**	**Summary of changes**
1.0	24.10.2013	Athanasios Mantes (TALENT)	Initial version
1.1	28.10.2013	G. Kuhns, R. Ebendt (DLR)	Minor corrections

Approval		
	Name	**Date**
Prepared	Manolis Koutlis	24.10.2013
Reviewed	All partners	24.10.2013-28.10.2013
Authorized	R. Ebendt (DLR)	28.10.2013

Circulation	
Recipient	**Date of submission**
European Commission	28.10.2013

Table of Contents

1 Introduction

The purpose of SimpleFleet project is to make it easy for SMEs, both, from a technological and business perspective, to create (Mobile) Web-based fleet management applications. For this purpose, an API providing users the necessary access to traffic data and functionality is required. The TrafficAPI stands on the shoulders of the TrafficStore; it uses data provided by the TrafficStore to provide the clients with functionality.

In that sense, the TrafficAPI plays a dual role. It provides users and clients of the SimpleFleet services access to the TrafficStore data, and it exposes functionality that is deemed necessary for common operations the users of the SimpleFleet services will need to perform. In other words, in the case of a SimpleFleet services integrator, it stands between the application the integrator develops, and the SimpleFleet data pool the TrafficStore provides.

This document serves two purposes: the first purpose is to report all functionality that has been added to the TrafficAPI the months that followed its initial launch (May 2013). Some debugging and testing took place, an authentication mechanism for controlling access to the API services was added, but, most importantly, functionality related to fleet management data was included. This was critical in the sense that it brought the TrafficAPI a lot closer to its final purpose: serving fleet managers with functionality using all acquired data and traffic analysis performed by mechanisms of the TrafficStore.

The second purpose, which is actually the main subject for this deliverable, is to describe the implementation of a bunch of “half-baked” apps, some incomplete pieces of software that demonstrate the use of the TrafficAPI and TrafficSDK implementations and can easily be adapted to existing applications or evolve without much effort into full scale fleet management applications. So, 9 simple applications were developed, in various programming languages and application environments in order to point out the usage of the TrafficAPI/TrafficSDK libraries and functionality and to demonstrate that choosing a REST architecture for delivering data and functionality was actually a wise choice, both from a developer’s view and from an integrator’s view.

Figure 1 relates the current deliverable to the rest of core technical deliverables. Deliverable D1.1 provided information about the data sources used in the SimpleFleet project. Deliverable D1.2 (TrafficStore) provides the glue that links data collection, map-matching (described in the D2.2 deliverable) and travel time aggregation and speed-profile computation together. Several services are to be built on top of the TrafficStore such as Time-parametrized shortest-path computation (Deliverable D3.1), Business Intelligence (Deliverable D3.2), Data visualization techniques (Deliverable D3.3), TrafficAPI and TrafficSDK deliverable (Deliverable D4.1). The current deliverable (Deliverable D4.2) is highlighted in red (depending from the TrafficAPI/TrafficSDK).

The outline of this deliverable is as follows: Section 2 argues about the TrafficAPI design decisions made and about the reason the TrafficAPI is a REST web services based API and for using XML and JSON formatted data. Section 3 describes the TrafficAPI in detail, starting from the data model objects and advancing to resources and mount points currently provided by the API. Some usage examples are also provided. Section 4 talks about the implementation; all involved technologies and frameworks are described in good detail, arguing about the reasons they were chosen. It also contains some details about the TrafficAPI documentation process and discusses the subject of the Traffic SDK. Finally, Section 5 contains some wrap-up conclusions about the TrafficAPI development so far and discusses future tasks and procedures. Section 6 contains some references about everything discussed in this deliverable.

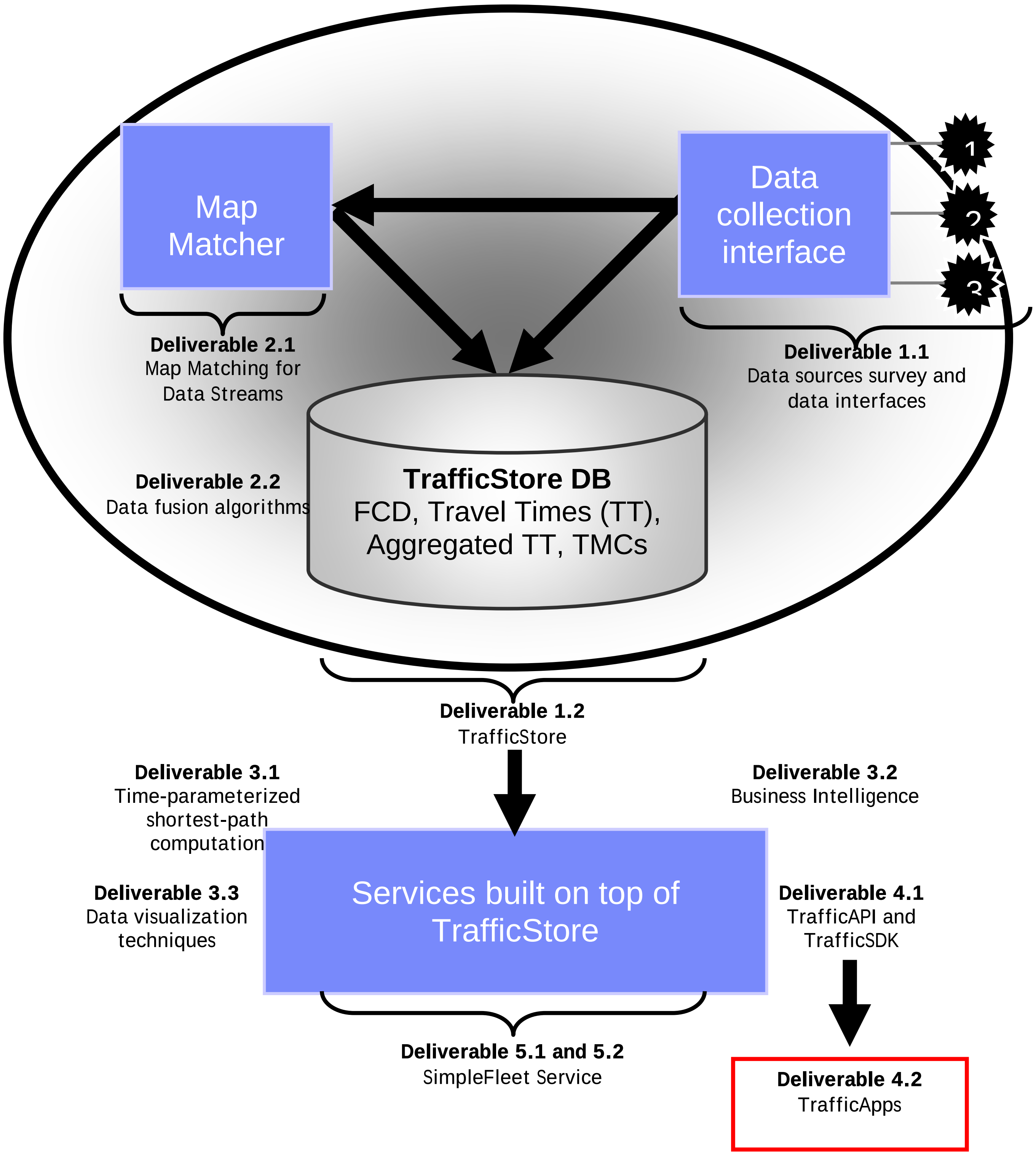


Figure 1: Core technical deliverables overview and integration

2 Features added to TrafficAPI

In the months following the initial TrafficAPI service launch, some much anticipated features were added to the TrafficAPI. In short, these include:

Authentication controlling mechanism

In all TrafficAPI HTTP GET calls, an extra parameter the `key=<API_key>` was added. This was nessesary in order to control access to functionality and to allow fleet managers to have access to data related only to their fleet. The key parameter is mandatory. For the time being, there is a Guest API key, which uses the key "**Gu3sT**"=(guest). This is just for testing and evaluation reasons, and in the future it will be blocked.

Routing and isochrones

So far, routing and isochrones functionality was available only through TrafficStore services. This has now changed, and the routing and isochrones functionality is now available through the TrafficAPI, from two different resource entry point groups:

- RoutingResource **(/routing/{city})**
- IsochronesResource **(/isochrones/{city})**

The parameters used by those HTTP GET requests were not changed in order to help all those using those services to transition easily from TrafficStore services calls to TrafficAPI services calls. Results from these service calls are only in JSON format. More details can always be found in the Enunciate generated documentation.

(http://snf-23150.vm.okeanos.grnet.gr:8090/simplefleet/docs/enunciate/).

Fleet related functionality

Using data and mechanisms developed from partners from DLR and the TrafficStore, the TrafficAPI was significantly enhanced (and it continues to add new features) with fleet management data and functionality. A new entry point resource, the *FleetResource* was added. A set of the main service calls added is the following:

1. **/fleet/{city}/{companyID}/currentlystopped**
2. **fleet/{city}/{companyID}/fuel**
3. **/fleet/{city}/{companyID}/status**
4. **/fleet/{city}/{companyID}/stops**
5. **/fleet/{city}/{companyID}/vehicledata**
6. **/fleet/{city}/{companyID}/{vehicleID}/fuel**
7. **/fleet/{city}/{companyID}/{vehicleID}/status**
8. **/fleet/{city}/{companyID}/{vehicleID}/stops**
9. **/fleet/{city}/{companyID}/{vehicleID}/vehicledata**

10. **/fleet/{city}/{companyID}/type/{vehicleTypeID}/status**

11. **/fleet/{city}/{companyID}/type/{vehicleTypeID}/vehicledata**

These services provide the fleet's currently stopped vehicles (1), fuel consumption data for entire fleets and individual vehicles on various dates (daily/weekly/monthly) (2, 6), location on entire fleets or individual vehicles (3, 7), stops statistics on fleets and vehicles (4, 8) and otjher vehicle data based on type of vehicle or not (5, 9, 10, 11). For all these details can be found in the documentation.

Together with these services, a new set of entity objects was implemented. The main objects used were three:

- *VehicleStop*, describing the data related to a stop of a vehicle
- *VehicleData,* wrapping all data related to a vehicle (GPS received data, ID, company ID, type, etc)
- *Vehicle LocationEpoch,* which includes a triplet of values, x, y, and timestamp, effectively describing where was a vehicle at a time instant.

These entity objects (especially the V*ehicleData* entity) will find usage in the following sample apps, where they will be described in greater detail. Again, full description about these entity properties can be found at the Enunciate generated documentation.

Spring framework usage

Finally, in order to facilitate easy expansion of the TrafficAPI services and internal functionality we included Spring framework on the configuration part of the CityRepository modules the TrafficAPI is using. This need stemmed from the fact that the routing and isochrone TrafficAPI services actually use calls to the respective TrafficStore services, and as such, a configuration url should be included for every CityRepository module. So we decided to add Spring in order to make such future expansions easier instead of re-writing the module configuration code each time. This is a sample of the Vienna datasource-cityrepository configuration Spring beans:

```
<bean id="viennaDataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource" singleton
= "true">
       <property name="url"><value>dburl</value></property>
       <property name="username"><value>user</value></property>
       <property name="password"><value>pass</value></property>
</bean>
<bean id="cityRepositoryVienna" class="gr.talent.simplefleet.CityRepository"
singleton = "true">
       <property name="dataSource">
         <ref bean="viennaDataSource"/>
    </property>
    <property name="cityName"><value>vienna</value></property>
    <property name="tablePrefix"><value>wien</value></property>
    <property name="epsg"><value>EPSG:32633</value></property>
    <property name="mapservicesURL"><value>http://snf-
14954.vm.okeanos.grnet.gr:10080/IsochronesServlet/Isochrones</value></property
>
</bean>
```

3 Fleet data

For testing and implementation of the new TrafficAPI services regarding fleet functionality the TrafficStore tables regarding the FleetAnalytics Input Data Model (as extensively described in Deliverable 3.2) were used. The main source of data was the B-K Telematics company for fleets that exist in Athens. As for the location of the test case, the consortium had decided to choose Athena, Greece, and to use a sub-fleet of B-K Telematics as a demonstration case. The main reason for this is that the Berlin and Vienna taxi fleets anonymize their FCD themselves, which makes it harder to track the vehicles of a closed user group. This problem is non-existent for the aforementioned fleet in Athens, since they do not apply an anonymization of the data themselves.

Deliverable 3.2 argues that, "in the scope of Fleet Analytics computation, apart from location and time information that is typically used in the scope of map-matching and other applications, the computation depends on the availability of vehicle engine on/off status, speed and vehicle type information". This kind of data is available for one sample fleet, provided by B-K Telematics for the city of Athens. This sample fleet contains ~400 distinct vehicle-IDs. B-K Telematics provides engine status, speed and detailed vehicle type information for this sample fleet and some additional information as well, e.g. vehicle run status (idle or not).

Three main tables from the TrafficStore were used:

- `fa_table_fleet_stops_<date>`
- `input_fcd_fleet_<date>`
- `speed_bins_<date>`

The first table contains data about vehicle stops (timestamps and positions). The second one contains all vehicle data received each time a vehicle telemetry unit sends its position to the TrafficStore (location, timestamp, speed, direction, etc). The third table contains information about the speed profile of the vehicles, the fuel consumption and the travelled distance.

All three tables contain a standard suffix that is related to the date of the data they contain. This is of essential meaning to the TrafficAPI services because the date is in most cases a mandatory parameter in providing data. So, the TrafficAPI services based on the date query parameter the HTTP GET usually contains are able to find and query the necessary table (or tables) for providing fleet related functionality.

4 TrafficApps

In the previous section some of the details around collection and processing of the fleet data that are related to possible fleet applications functionality were revisited. In the following section, we will discuss 9 sample incomplete pieces of software that with minimum effort can be extended to full programs, or be part to already existing software solutions as added functionality. These small programs/code snippets were implemented in various programming languages or application programming frameworks in order to demonstrate how easy it is for a developer to implement the TrafficAPI/TrafficSDK functionality in most common application development scenarios, and to confirm that the REST API/ CityRepository architecture implementation chosen for the TrafficAPI/TrafficSDK is functional and effective.

4.1 *Where are our fleet's vehicles?*

Task Description

The most common task of every fleet management software is the actual position of the fleet's vehicles at any given time. This sample app addresses the task of displaying all fleet vehicle positions on a map viewed by a web.

Implementation

For this sample app we will use Javascript and the OpenLayers web mapping framework. We will also use jQuery, a Javascript API that can address issues such as JSON data parsing or HTTP requests.

TrafficAPI call

http://<server_path>/rest/fleet/<city>/<company_id>/status?key=<api_key>

Development details

A sample app was developed in javascript using Openlayers to just show the locations of the fleets vehicles on the map.

The main TrafficAPI/TrafficSDK call used for this sample app was the following:

http://snf-23150.vm.okeanos.grnet.gr:8090/simplefleet…

…/rest/fleet/athens/78/status?key=Gu3sT

This HTTP GET request can is used for fetching data from athens, for fleet with id=78 that contain information about the current status of the fleet's vehicles. One can assume that the data fetched should contain the current positions of the fleet's vehicles, which is the information we need for this app. Note that vehicle data fetched by this call contain a lot more information, such as speed, direction, fuel consumption etc. More details can be found in Deliverable 3.2 of the project.

Because this is our first example app (and we're going to use vehicle-related data a lot in the following apps), this is an example of an array of VehicleData objects in JSON representation:

```
{
      "vehicle_data": [
            {
                  "companyID": "78",
                  "vehicleID": "40",
                  "vehicleTypeID": "5",
                  "x": "23.755567",
                  "y": "37.940017",
                  "timestampGPS": "1382352522000",
                  "timestampInsertTime": "1382352900000",
                  "speed": "75.0",
                  "direction": "352.0",
                  "altitude": "0",
                  "accuracy": "0",
                  "fuelConsumption": "0.0",
                  "RFUCellID": "0",
                  "engineFlag": "false",
                  "idleFlag": "false",
                  "SRID_GPS": "4326",
                  "timezone": "+00:00"
            },
            ...
      ]
}
```

One can observe the information contained in the VehicleData objects. These objects are directly mapped to the records contained in the **input_fcd_data** tables of the TrafficStore. TrafficStore related deliverables mention that for each day there is a different table in the TrafficStore that contains data gathered from the various fleets at that specific day.

Also, since this is the first TrafficApps example using javascript, it is a proper spot to mention some implementation details that are related specifically with javascript webpage scripting.

The main detail that a developer cares about is how the TrafficAPI for SimpleFleet services will be invoked through Javascript. Although there are many ways of sending HTTP GET requests to a REST service through Javascript, we used jQuery. jQuery is a multi-browser (cf. cross-browser) JavaScript library designed to simplify the client-side scripting of HTML. Used by over 65% of the 10,000 most visited websites, jQuery is the most popular JavaScript library in use today. jQuery is free, open source software, licensed under the MIT License. jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications. jQuery also provides capabilities for developers to create plug-ins on top of the JavaScript library. This enables developers to create abstractions for low-level interaction and animation, advanced effects and high-level, theme-able widgets. The modular approach to the jQuery library allows the creation of powerful dynamic web pages and web applications.

So, at first we created a very simple “HelloWorld” sample OpenLayers map webpage, and we included the nessesary jQuery scripts. Then, a javascript method to perform an HTTP GET request to the TrafficAPI service was implemented:

```
var response;

function requestToTrafficAPI(){

var req = $.ajax({
url: (insert TrafficAPI GET URL here),
            type: "GET",
            dataType: "json"
      });

      req.done(function(message){
            response = message;
      });
}
```

This simple method is all we need to perform any HTTP GET request described to the TrafficAPI documentation. There are three main points that one should note:

- The $.ajax jQuery method takes care of the actual HTTP GET request. The ***url*** parameter is where a specific REST API url is inserted, in order to “ask” the TrafficAPI server for specific data.
- Note that the ***datatype*** parameter in the request is **“json”**. This means that we ask for data returned in JSON format. XML or any other format can be used, according th the valid response specifications of the TrafficAPI calls at each case.
- There is a `req.done` callback which conveniently allows us to handle returned data when they are ready. In most cases, returned data are stored to an external variable (in our code snippet this is the `response` variable) for further processing.

Finally, we process the response of this method in order to insert markers on our OpenLayers map. This is done with this small code snippet:

```
// Variable data contains the data we get from serverside
success: function(data) {
      var i=0;
      for (i=0; i<data["vehicle_data"].length; i++) {
            var o = data["vehicle_data"][i];

            var loc1 = new OpenLayers.Geometry.Point(o["x"], o["y"]);
            var loc2 = loc1.transform(fromProjection,toProjection);

            var feature = new OpenLayers.Feature.Vector(
                  loc2,

      {some:'data',id:o["vehicleID"],type:o["vehicleTypeID"]},
                  {externalGraphic: 'img/Marker-Chartreuse.png',
                        graphicHeight: 48, graphicWidth: 48});
            vectorLayer.addFeatures(feature);
      }
}
```

This code parses the JSON data returned by the request, and for each VehicleData object it creates a point on the map displayed by a marker icon. Note that VehicleData objects contain geometry in WGS84 datum and in lat-long projection, so, before adding the points on the map, we do a transformation of their coordinates.

This is an example of a very small application that with just a few lines of code can perform a critical task for a fleet management business. The key to this example (and all examples that will follow lies on three things:

1. How to perform a request for data to the TrafficAPI server
2. How to use the TrafficSDK objects and routines to parse the request's response
3. How to handle the data in order to be useful for the task that one wants to achieve.

The final result of this sample app can be seen on the following image:

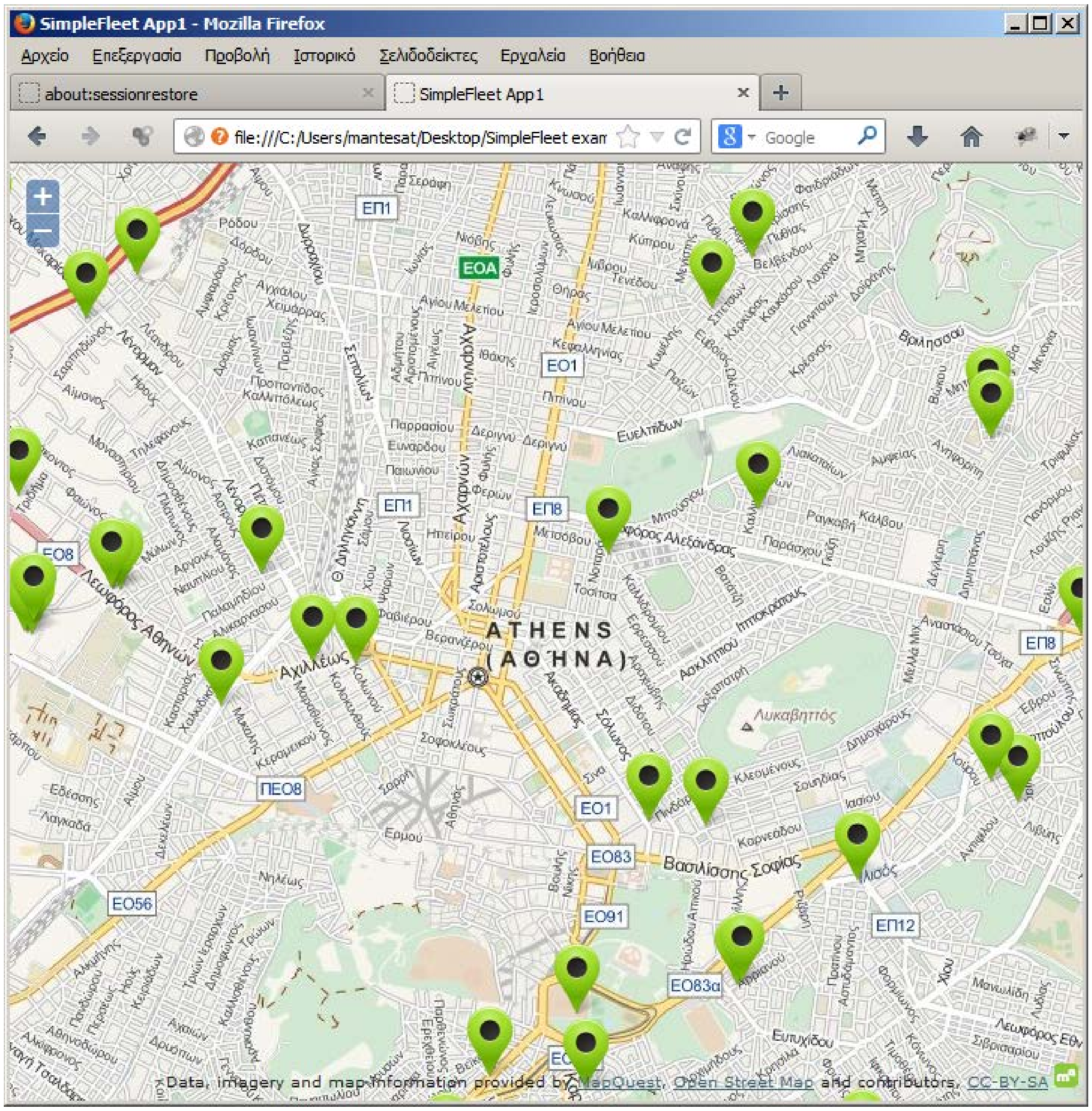

SimpleFleet App1 - Mozilla Firefox
Αρχείο Επεξεργασία Προβολή Ιστορικό Σελιδοδείκτες Εργαλεία Βοήθεια
about:sessionrestore
SimpleFleet App1
file:///C:/Users/mantesat/Desktop/SimpleFleet exam
Google
ATHENS (ΑΘΗΝΑ)
Λυκαβηττός
Data, imagery and map information provided by MapQuest, Open Street Map and contributors, CC-BY-SA

***Image 1**: Location of fleet's vehicles on a web map*

4.2 Which is the closest vehicle to a point?

Task Description

Find the fleet's vehicle closest to a point (in order to reach it and provide some kind of service).

Implementation

For this sample app we will use Java together with the Jersey REST application framework helper classes, for retrieving and parsing JSON data on the client side.

TrafficAPI call

http://<server_path>/rest/fleet/<city>/<company_id>/status?key=<api_key>

Development details

By using the same resource entry point as before we can also answer another fundamental task: Find which vehicle is closest to a location. We argued about the use of Jersey framework at the server, in Deliverable 4.1 about the TrafficAPI and SDK design and implementation. Now, we will use Jersey on the client side.

In Deliverable 4.1 we also described the process of using the Enunciate framework to scrape client libraries and documentation out of existing REST services APIs. Here we took advantage of this process to produce the SDK wrapper objects for java. This is how the VehicleData class was automatically produced.

The listing of this small java snippet's main() method is the following:

```
public static void main(String[] args){

    double pointLon = 23.7;
    double pointLat = 38;
    String baseURL = "http://snf-
23150.vm.okeanos.grnet.gr:8090/simplefleet/rest";

    Client client = Client.create();
        ParameterizedType genericType = new ParameterizedType() {
            @Override
            public Type[] getActualTypeArguments() {
                return new Type[] {VehicleData.class};
            }
            @Override
            public Type getRawType() {
                return List.class;
            }
            @Override
            public Type getOwnerType() {
                return List.class;
            }
        };
        GenericType<List<VehicleData>> type
            = new GenericType<List< VehicleData >>(genericType) {};
        List<VehicleData> result = client.resource(baseURL+
"/fleet/athens/78/status?key=Gu3sT").get(type);
```

```
        VehicleData closest = null;
        double distance = Double.MAX_VALUE;
        for (int i=0;i<result.size();i++){
            VehicleData candidate = result.get(i);
            double d = Math.sqrt(Math.pow((candidate.getX()-
pointLon),2)+
                                 Math.pow((candidate.getY()-
pointLat),2));
            if (d<distance){
                distance = d;
                closest = candidate;
            }
        }

        System.out.println("Closest vehicle is
:"+closest.getVehicleID());
    }
}
```

A small walk through this code:

1. We‘re using Jersey’s helper class Client to perform the request to the server. This hides out a lot of HTTP GET request details and makes tha task of fetching data easy.

2. We use a small utility class, the `ParameterizedType`, in order to create some lists of generic types. This solves the problem of directly inserting the response objects into lists of the desired object type. In our case, we use this class to create lists of VehicleData objects directly from the response data.

3. Note that the VehicleData class was (together with other entity classes) automatically scraped off from the TrafficAPI code by Enunciate.

4. Finally, we’re performing a primitive distance computation in order to find the closest (by terms of Euclidean distance) vehicle to a given point.

This example clearly shows that in a Java application development environment it is relatively easy to get data and functionality provided by the TrafficAPI/SDK into the developer’s code. One could argue that the task’s target is not achieved in terms of realism since in real life, when calculating the closest vehicle to a point one should compute the routing distance between the vehicles and the point, instead of the Euclidean one. Although this is absolutely correct, for the purpose of this example it is just a detail: the purpose here was to show that including the TrafficAPI data and functionality in a java application development environment is a trivial task. Of course, a developer can expand the above example as he pleases, calculating either the absolute road distance between a point and a vehicle, or the smallest travel time distance. After all, both these cases are already covered by the routing functionality provided by the TrafficAPI.

4.3 Where and when did this vehicle stop?

Task Description

A common task of fleet management software is to identify the stops that a vehicle made during a working day.

Implementation

For this sample app we will use PHP.

TrafficAPI call

```
http://<server_path>/rest/fleet/<city>/<company_id>/<vehicle_id>/stops/
status?key=<api_key>&date=<date_string>
```

Development details

This task can be performed on the server side of a fleet management software, and it clearly demonstrates the argument that the TrafficAPI should be implemented in a group of REST web services. The php code for this task is very simple:

```
<?php
        $json =json_decode(
file_get_contents("http://snf-
23150.vm.okeanos.grnet.gr:8090/simplefleet/rest/fleet/athens/78/80720/s
tops?key=Gu3sT&xml=false&date=y2013m10d01"));
        foreach ($json->{'v_stop'} as $key=>$val ) {
                echo "<tr><td>".$val->vehicleID."</th>"
                        . "<td>" . $val->vehicleTypeID . "</th>"
                        . "<td>" . $val->startX  . "</th>"
                        . "<td>" . $val->startY . "</th>"
                        . "<td>" . $val->endX . "</th>"
                        . "<td>" . $val->endY . "</th>"
                        . "<td>" . $val->startTimestamp . "</th>"
                        . "<td>" . $val->endTimestamp  . "</th></tr>"
                        . "\r\n";
        }
?>
```

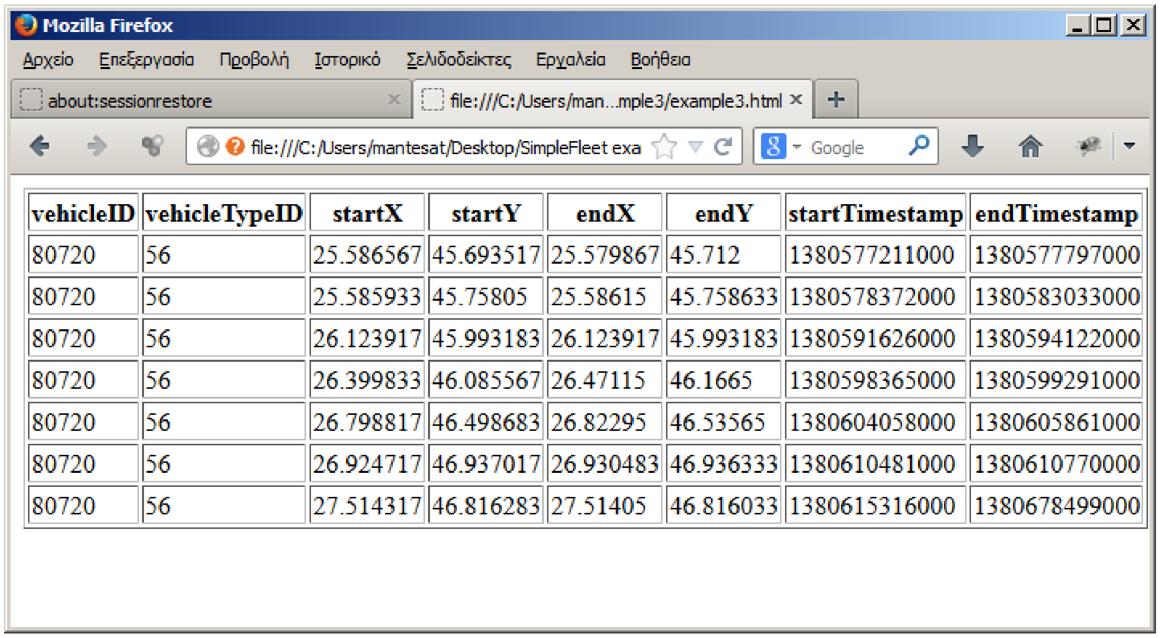
Note that this task should it be written in an application programming language such as Java or C#, it would use another entity, the VehicleStop entity, which wraps all data provided by the TrafficStore about vehicle stops. However, this is not the case here. In PHP there are libraries already included that can parse the JSON response directly.

So, in our example, the two basic steps of requesting the data and parsing the response are performed in just one line of code (although a little long line, but that's because of the TrafficAPI url ☺):

1. The function `file_get_contents(url)` performs the task of the HTTP GET request to the TrafficAPI server, and

2. `The function json_decode(result)` decodes the response to a json object

The result of these calls is finally formatted to an HTML webpage form to present the results in tabular form:


Mozilla Firefox
Αρχείο Επεξεργασία Προβολή Ιστορικό Σελιδοδείκτες Εργαλεία Βοήθεια
about:sessionrestore
file:///C:/Users/man...mple3/example3.html
file:///C:/Users/mantesat/Desktop/SimpleFleet exa
Google

vehicleID	vehicleTypeID	startX	startY	endX	endY	startTimestamp	endTimestamp
80720	56	25.586567	45.693517	25.579867	45.712	1380577211000	1380577797000
80720	56	25.585933	45.75805	25.58615	45.758633	1380578372000	1380583033000
80720	56	26.123917	45.993183	26.123917	45.993183	1380591626000	1380594122000
80720	56	26.399833	46.085567	26.47115	46.1665	1380598365000	1380599291000
80720	56	26.798817	46.498683	26.82295	46.53565	1380604058000	1380605861000
80720	56	26.924717	46.937017	26.930483	46.936333	1380610481000	1380610770000
80720	56	27.514317	46.816283	27.51405	46.816033	1380615316000	1380678499000

Image 2*: The PHP formatted output as an HTML webpage*

4.4 How many stops did our fleet's vehicles do last week?

Task Description

This sample app addresses the task of producing a report of vehicle stops for a given interval.

Implementation

For this sample app we will use Java, coupled with Jersey again. The new thing here is that we are going to use the Apache POI framework in order to automatically produce some .xls spreadsheets with the necessary data.

TrafficAPI call

```
http://<server_path>/rest/fleet/<city>/<company_id>/
stops/status?key=<api_key>&date=<date_string>
```

Implementation details

The sample app described in section 4.2 has already made clear the way a developer can use Java and jersey libraries in order to process REST HTTP GET responses in JSON format. Here we will just add the code snippet that is necessary for producing .xls files using the Apache POI framework. Assuming that we're catching up from a modified version of the code snippet in section 4.2 (so as to use 7 consecutive HTTP GET calls in order to return VehicleStop data instead of VehicleData objects for all days of a week), we reach a point where we have either a combined list of VehicleStop objects for all week, or 7 VehicleStop lists for each day of the week. The following code snippet creates a sheet on a spreadsheet file:

```
ArrayList<VehicleStop>[] stopsPerDay = new ArrayList[7];
// Add data to lists from responses...
// ...
// Use Apache POI to populate the .xls file
Workbook wb = new HSSFWorkbook();
DateFormatSymbols symbols = new DateFormatSymbols(new Locale("gr"));

String[] dayNames = symbols.getShortWeekdays();
for (String s : dayNames) {
    System.out.print(s + " ");
}

for (int i=0;i<7;i++){
    Sheet sheet = wb.createSheet(dayNames[i]);
    Row headers = sheet.createRow(0);
    headers.createCell(0).setCellValue("VehicleID");
    headers.createCell(1).setCellValue("TypeID");
    headers.createCell(2).setCellValue("Stops");

      // some cell shading code…

}
```

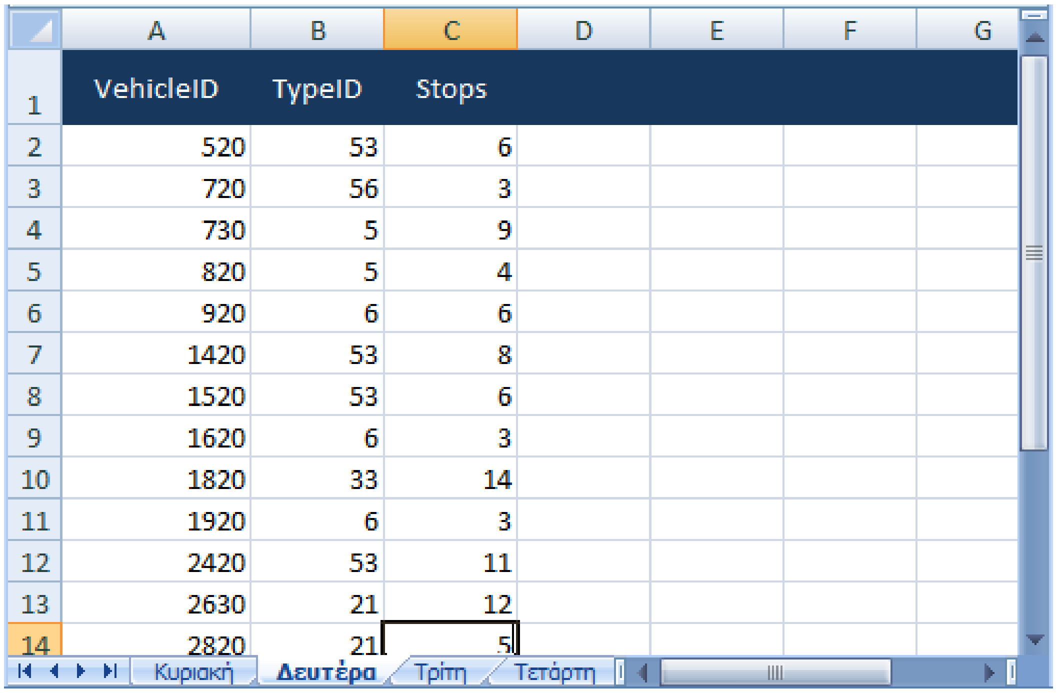
So far we have created seven sheets, each with a day's name. Now we will populate them with data:

```
for (int i=0;i<7;i++){
    Sheet sheet = wb.getSheetAt(i);
    int vehicleID=-1;
    int vehicleType=-1;
    int stopsSum =0;
    int vehiclesCount=0;

    for (int j=0;j<stopsPerDay[i].size();j++){
        VehicleStop stop = stopsPerDay[i].get(j-1);
        int currentVehicleID =stop.getVehicleID();
        int currentVehicleType =stop.getVehicleTypeID();
        if (currentVehicleID!=vehicleID){
            if (vehicleID!=-1){
                Row data = sheet.createRow(vehiclesCount++);
                data.createCell(0).setCellValue(vehicleID);
                data.createCell(1).setCellValue(currentVehicleType);
                data.createCell(2).setCellValue(stopsSum);
                vehicleID = currentVehicleID;
                vehicleType = currentVehicleType;
            }
        }else{
            stopsSum+=1;
        }
    }
      //write the last vehicle row when finished

}
```

This code traverses all VehicleStops and adds up stops of the same vehicle in order to create a row with the sum of all stops during that day of the week. This is a screenshot (a little shrinked) of the final result:



	A	B	C	D	E	F	G
1	VehicleID	TypeID	Stops				
2	520	53	6				
3	720	56	3				
4	730	5	9				
5	820	5	4				
6	920	6	6				
7	1420	53	8				
8	1520	53	6				
9	1620	6	3				
10	1820	33	14				
11	1920	6	3				
12	2420	53	11				
13	2630	21	12				
14	2820	21	5				

***Image 4**: Vehicle on an OpenStreetMap on iPhone, showing its fuel consumption in a label.*

4.5 *Who’s stopped right now?*

Task Description

This sample app addresses the task of finding which vehicles are currently stopped. It produces a list of stopped vehicles together with the time the vehicles are not moving. This report is produced in a simple textual format. Note that according to Fleet Management related deliverables, a “stop” of a vehicle is defined by a pair of timestamps, the “start_t” timestamp and the “end_t” timestamp. So, when a vehicle has a start_t timestamp but not an “end_t” timestamp, this means that it has stopped.

Implementation

For this sample app we will use Java again. This time, we will not use Jersey or any other external libraries, but we will rely on the XML parsing functionality already incorporated into java with the JAXB framework. This means that all requests will use XML as an output format.

TrafficAPI call

```
http://<server_path>/rest/fleet/<city>/<company_id>/currentlystopped/st
atus?key=<api_key>
```

Implementation details

This is a small main() method that performs this tack in Java:

```java
public static void main(String[] args){
    String baseURL = "http://127.0.0.1:8080/SimpleFleet/rest/fleet";

    java.net.URL url;
    try {
        url = new java.net.URL(baseURL +
            "/athens/78/stops?key=Gu3sT&xml=true&date=y2013m10d01");
        JAXBContext jc = JAXBContext.newInstance(Wrapper.class,
                    VehicleStop.class);
        java.net.URLConnection connection = url.openConnection();
        connection.connect();
        Unmarshaller unmarshaller = jc.createUnmarshaller();

        List<VehicleStop> results = unmarshal(unmarshaller,
                VehicleStop.class, connection.getInputStream());

        //handle the result as needed...

        System.out.println("Total Stops :"+results.size());

        System.out.println("vid\tvtid\tcid\tx\ty\tduration");
        DecimalFormat df = new DecimalFormat( "##0.0000" );
        for (int i=0;i<100;i++){
            VehicleStop stop = results.get(i);
            long millis = stop.getEndTimestamp()
                          -stop.getStartTimestamp();
            System.out.println(
                stop.getVehicleID()+"\t"+
                stop.getVehicleTypeID()+"\t"+
                stop.getCompanyID()+"\t"+
                df.format(stop.getEndX())+"\t"+
                df.format(stop.getEndY())+"\t"+
                String.format("%d min, %d sec",
                        TimeUnit.MILLISECONDS.toMinutes(millis),
    TimeUnit.MILLISECONDS.toSeconds(millis) -
    TimeUnit.MINUTES.toSeconds(
TimeUnit.MILLISECONDS.toMinutes(millis))));
        }
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (JAXBException e) {
        e.printStackTrace();
    }
}
```

There are some important points that worth noticing:

- We are using a *VehicleStops* class to handle deserialization of the XML response from the initial HTTP GET request. This is a class automatically generated by Enunciate for the TrafficSDK java flavor.
- A small *Wrapper* class is used for deserializing list results.

- JAXB deserialization process is almost as simple as the Jersey one we saw on a previous Java example app.

The output of this sample app is a tab delimited text. Note that this can easily be imported into spreadsheet processing software:

```
                    Total Stops :2558
vid     vtid    cid     x           y           duration
520     53      78      23,7724     37,8526     20 min, 5 sec
520     53      78      23,7610     37,8539     7 min, 25 sec
520     53      78      23,7635     37,8537     10 min, 0 sec
520     53      78      23,7500     37,8642     145 min, 17 sec
520     53      78      23,7684     37,8605     3 min, 40 sec
520     53      78      23,8569     38,1718     674 min, 6 sec
520     53      78      23,8556     38,1732     339 min, 56 sec
720     56      78      24,9869     45,8405     3 min, 51 sec
720     56      78      25,5851     45,7581     376 min, 47 sec
720     56      78      25,5863     45,7584     394 min, 28 sec
730     5       78      27,9193     36,3371     48 min, 30 sec
730     5       78      27,8096     36,2637     14 min, 30 sec
730     5       78      27,8344     36,2536     9 min, 50 sec
730     5       78      27,8589     36,2286     21 min, 55 sec
730     5       78      27,8536     36,2268     31 min, 50 sec
730     5       78      27,7784     36,1536     39 min, 51 sec
730     5       78      27,8177     36,0627     16 min, 10 sec

...
```

4.6 What’s the ETA for this route?

Task Description

A quite common task for time critical fleet services is to provide accurate arrival time (or ETA, estimated time of arrival). So, it is not only necessary to know the fastest route between two or three points, but also the travel time between those points.

Implementation

For this sample app we will use C#. We will make use of the TrafficSDK client libraries automatically produced by Enunciate for C#, and we will use the standard routing functionality offered by the TrafficAPI.

TrafficAPI call

The TrafficAPI call that will be used is:

```
http://<server_path>/rest/routing/<city>/?x1=<x1>?y1=<y1>?x2=<x2>?y2=<y
2>?key=<api_key>
```

Implementation details

Since this is the first example using C#, as in any other case with other application development frameworks or programming languages, we must somehow make the HTTP GET call to the TrafficAPI server, and process the response results. To perform the HTTP GET request, we use the following code snippet:

```
public string MakeRequest(string parameters) {
    var request = (HttpWebRequest)WebRequest.Create(EndPoint +
parameters);

    request.Method = Method.ToString();
    request.ContentLength = 0;
    request.ContentType = ContentType;

    if (!string.IsNullOrEmpty(PostData) && Method == HttpVerb.POST) {
        var encoding = new UTF8Encoding();
        var bytes = Encoding.GetEncoding("UTF-8").GetBytes(PostData);
        request.ContentLength = bytes.Length;
        using (var writeStream = request.GetRequestStream()){
            writeStream.Write(bytes, 0, bytes.Length);
        }
    }

    using (var response = (HttpWebResponse)request.GetResponse()){
        var responseValue = string.Empty;
        if (response.StatusCode != HttpStatusCode.OK) {
            var message =
String.Format("Request failed. Received HTTP {0}",
response.StatusCode);
            throw new ApplicationException(message);
        }
        // grab the response
        using (var respStream = response.GetResponseStream()){
            if (responseStream != null)
                using (var reader = new StreamReader(respeStream)) {
                    responseValue = reader.ReadToEnd();
                }
        }
        return responseValue;
    }
}
```

Using this method we can get responses in a plain string form from any TrafficAPI request. For convenience, we can wrap such utility methods into a helper class, for example a *TrafficAPIClient* class.

By using the TrafficSDK generated by Enunciate for C#, we can transform responces either into c# objects, or directly parse them. In our case, for simplicity, we will directly parse the JSON response:

```
public static void Main (string[] args){
    var client = new TrafficAPIClient();
    client.EndPoint = @"http://snf-
23150.vm.okeanos.grnet.gr:8090/simplefleet/rest/routing/vienna/?x1=16.3
43536&y1=48.223548&x2=16.402588&y2=48.246759&key=Gu3sT";
    client.Method = HttpVerb.GET;
    var json = client.MakeRequest("");
    string[] parts = s.Split(,);
    foreach (string part in parts)
    {
        part = part.Trim();
        if (part.StartsWith("\"tt\""){
            string[] split2 = part.Split(":");
            Console.WriteLine(
"Travel time between these points is:"+
        TimeSpan.FromMilliseconds(
        (long) Convert.ToDouble(split2[1].Trim()));
        }
    }
}
```

The above snippet just prints the travel time of the current shortest path between two points on a city's dataset:

```
Travel time between these points is: 0:17:36.7900000
```

It is obvious that the example can be more elaborate in terms of error checking, result formatting, etc, but this is not the purpose of this example. We kept the example as simple as possible in order to show how one can use the TrafficAPI/SDK functionality together with C#.

Note that as we mentioned above, the same thing could be achieved by helper classes generated by Enunciate and a proprietary JSON parsing library for C#. All helper classes can be found in the TrafficAPI/SDK documentation page.

4.7 How much fuel did a vehicle burn so far today?

Task Description

A quite common task for time critical fleet services is to provide accurate arrival time (or ETA, estimated time of arrival). So, it is not only necessary to know the fastest route between two or three points, but also the travel time between those points.

Implementation

For this sample app we will use Objective-C and Mac OS X's prime development environment, Xcode. We will make use of the TrafficSDK client libraries automatically produced by Enunciate for Objective, and we will use the standard vehicle data functionality offered by the TrafficAPI.

TrafficAPI call

```
http://<server_path>/rest/fleet/<city>/<company_id>/<vehicle_id>/fuel/s
tatus?key=<api_key>&date=<date_string>
```

Implementation details

As already mentioned in Deliverable 4.1 about the TrafficAPI/TrafficSDK, Enunciate created client libraries for Objective-C. These libraries contain the functionality for marshalling / unmarshalling XML and JSON responses to / from Objective-C objects. The main entity object we are using here is the *VehicleData* entity, which in Objective-C generated code is called *ENUNCIATENS0VehicleData.* The standard code for requesting data from the TrafficAPI service and transforming the response into an entity object is the following:

```
NSData *responseData;
//data holding the XML from the response.

NSURL *baseURL = @"http://snf-
23150.vm.okeanos.grnet.gr:8090/simplefleet/rest";
//the base url including the host and subpath.
NSURL *url = [NSURL URLWithString: @"http://snf-
23150.vm.okeanos.grnet.gr:8090/simplefleet/rest/fleet/athens/78/status?
key=Gu3sT"];

NSMutableURLRequest *request = [[NSMutableURLRequest alloc]
initWithURL:url];
NSURLResponse *response = nil;

NSError *error = NULL; [request setHTTPMethod: @"GET"];
//this example uses a synchronous request,
//but you'll probably want to use an asynchronous call
responseData = [NSURLConnection sendSynchronousRequest:request
      returningResponse:&response error:&error];
NSString *string = [[NSString alloc] initWithData:responseData
      encoding:NSUTF8StringEncoding];

ENUNCIATENS0VehicleData *responseElement = [ENUNCIATENS0VehicleData
      readFromXML: responseData];
```

Again, one can observe that it is quite easy to request data and parse the response, even in Objective-C, thanks to:

1. Enunciate generated classes for Objective –C (which make the TrafficSDK offered for this programming language, and,

2. The built-in mechanism in Objective-C to perform HTTP GET requests and handle responses.

This small Objective-C code chunk is actually part of the iFleet prototype that will be delivered at the end of the SimpleFleet project. The iFleet will be an IOS application running on iPhones / iPads which will hopefully include all the functionality described in all these sample app examples. It is out of the scope of this deliverable to include here the code for actually handling the *ENUNCIATENS0VehicleData* object received by the request in the above code snippet, but one can see an intermediate result of a Vehicle on an OpenStreetMap on the iPhone, with a Vehicle marker and a label indicating the current fuel consumption.

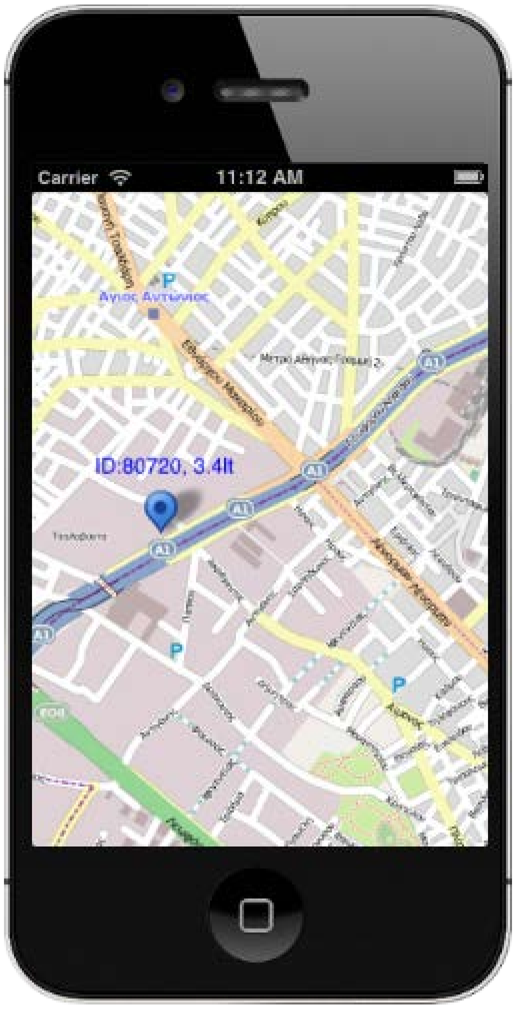


Image 3*: Vehicle on an OpenStreetMap on iPhone, showing its fuel consumption in a label.*

4.8 What’s the best route from A to B?

Task Description

A much needed task for real-time dispatching services is to provide accurate routing between a startup point and a destination. So, this sample app will demonstrate the visualization of the shortest path between two points, based on the current traffic for a city.

Implementation

For this sample app we will use Javascript, and jQuery. We will also use the standard routing functionality offered by the TrafficAPI.

TrafficAPI call

```
http://<server_path>/rest/routing/<city>/?x1=<x1>?y1=<y1>?x2=<x2>?y2=<y
2>?key=<api_key>
```

Implementation details

This sample application is essentially the same in it’s basic details with the first application (“Where are our vehicles now”) in section 3.1. However, there are some differences:

1. Instead of using an entity object from the TrafficSDK, we’re directly parsing the response (that of course means that we know the details of the JSON structure of the response).

2. We are placing a simple line showing the route on the map.

This is the code snippet that does it all:

```
// add encoded polyline from URL using Ajax
$(document).ready(function() {

      vectorLayer = new OpenLayers.Layer.Vector("line");
      map.addLayer(vectorLayer);

      $.ajax({ // ajax call starts
            url: 'http://snf-
23150.vm.okeanos.grnet.gr:8090/simplefleet/rest/routing/vienna/',
            data:
'x1=16.343536&y1=48.223548&x2=16.402588&y2=48.246759&x3=16.375980&y3=48
.176179&key=Gu3sT',
            // Send value of the clicked button
            dataType: 'json', // Choosing a JSON datatype
            cache: false,
            // Variable data contains the data we get from serverside
            success: function(data)
            {
                  var format = new OpenLayers.Format.EncodedPolyline(
{geometryType:"linestring"});
                  var encoded=data.path1_2;
                  var f = format.read(encoded);
                  f.geometry
= f.geometry.transform(fromProjection, toProjection);
                  f.style = {"strokeColor":'#0000FF', "strokeWidth":4};
```

```
                        vectorLayer.addFeatures(f);
                }
        });
});
```

As one can notice, it is easy to add lines to the map, using the Openlayers API.

The following inmage shows the final result:

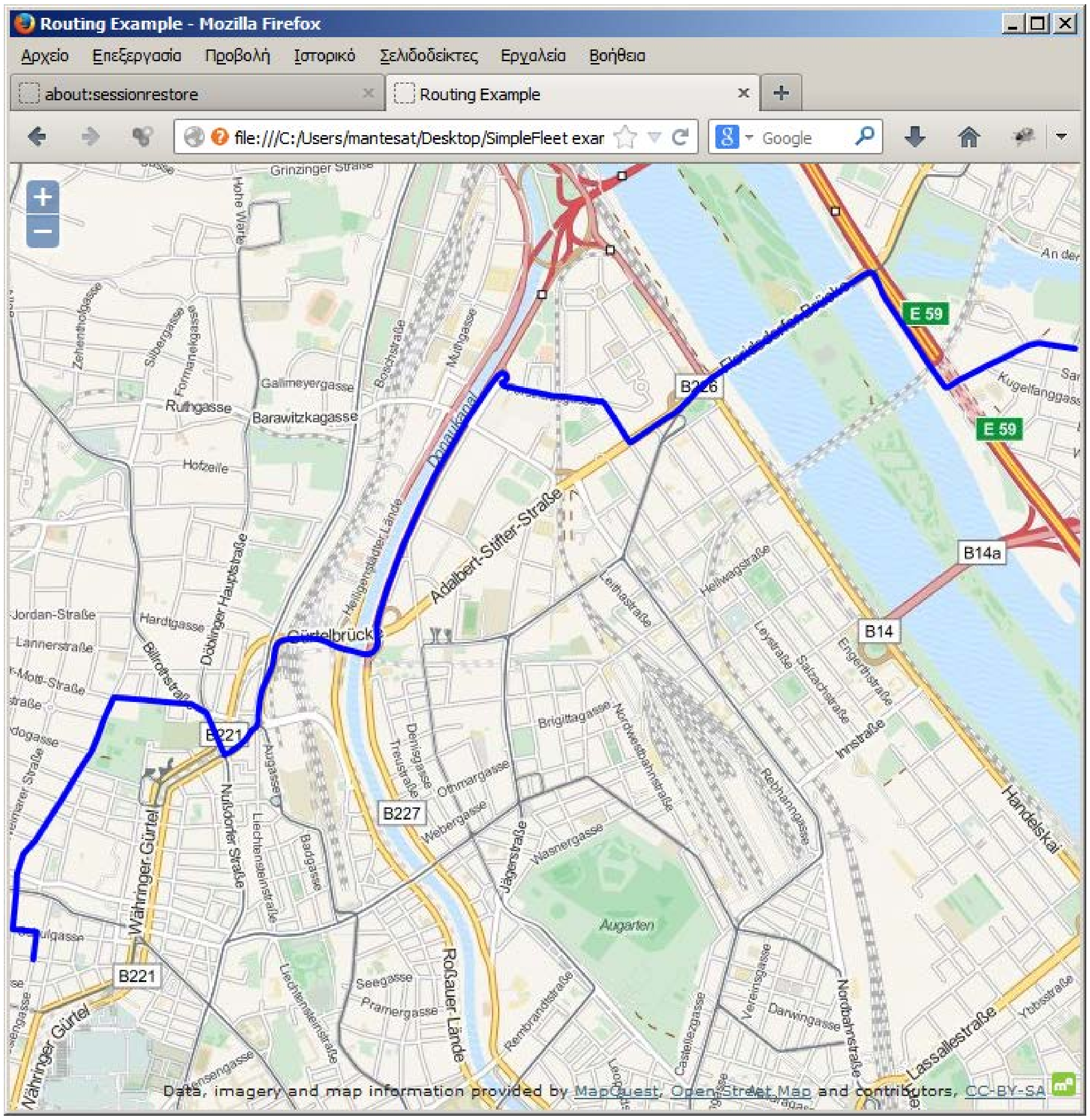

Routing Example - Mozilla Firefox
Αρχείο Επεξεργασία Προβολή Ιστορικό Σελιδοδείκτες Εργαλεία Βοήθεια
about:sessionrestore
Routing Example
file:///C:/Users/mantesat/Desktop/SimpleFleet exar
Google
Data, imagery and map information provided by MapQuest, Open Street Map and contributors, CC-BY-SA

***Image 4**: Routing between two points*

4.9 Show me isochrones on my map

Task Description

Isochrones can be a very good visualizing tool for strategic decisions about fleet management. This task is about adding isochrones to a GoogleMaps webpage.

Implementation

For this sample app we will use Javascript, jQuery, and GoogleMaps Javascript API. We will also use the standard isochrone functionality offered by the TrafficAPI.

TrafficAPI call

```
http://<server_path>/rest/isochrones/<city>?x=<x>&y=<y>&max_tt=<max_tt>
&num=<num>&key=<api_key>
```

Implementation details

This is the third sample app that it’s based on Javascript, so by now things on request and responce management of the TrafficAPI data and functionality should be straightforward. We‘re only going to show some details regarding the Google Maps API and its interaction with TrafficAPI fetched data:

Setting up a Google map using javascript and the Google Maps API is easy:

```
function initialize() {
    var myLatlng = new google.maps.LatLng(38, 23.7);
    var myOptions = {
        zoom: 12,
        center: myLatlng,
        mapTypeId: google.maps.MapTypeId.ROADMAP
    }
    map = new google.maps.Map(
            document.getElementById("map"), myOptions);
      restRequest();
      marker.setMap(map);

}
```

Note that at the very last line of the above code snippet, a marker is added to the map. This marker is defined like this:

```
var marker=new google.maps.Marker({
    position:myCenter,
    url: '/',
    animation:google.maps.Animation.BOUNCE
});
```

and in real-time display, it is bouncing ☺. We have implemented some added functionalities like moving the marker or placing it with a click, etc, but that’s actually trivial. The marker is used for defining a “center” for the isochrone polygons.

Finally, we add some isochrones to the map:

```
function restRequest(){
      var req = $.ajax({
            url: "http://snf-
23150.vm.okeanos.grnet.gr:8090/simplefleet/rest/isochrones/athens/?x=23
.727809143065946&y=37.984053332382516&max_tt=10000&num=3&key=Gu3sT",
            type: "GET",
            dataType: "json"
      });

      req.done(function(msg){
            response = msg;
            var decodedPath = google.maps.geometry.encoding.decodePath(
                  response.isochrone1);
            var decodedLevels = decodeLevels("");
            var setRegion1 = new google.maps.Polygon({
                  path: decodedPath,
                  levels: decodedLevels,
                  strokeColor: "#00FF00",
                  strokeOpacity: 1.0,
                  strokeWeight: 2,
                  fillColor: '#7F7F7F',
                  fillOpacity: 0.3,
                  map: map
            });
            var setRegion2 = new google.maps.Polygon({
                  path: google.maps.geometry.encoding.decodePath(
                        response.isochrone2),
                  levels: decodedLevels,
                  strokeColor: "#FFFF00",
                  strokeOpacity: 1.0,
                  strokeWeight: 2,
                  fillColor: '#7F7F7F',
                  fillOpacity: 0.3,
                  map: map
            });
            var setRegion3 = new google.maps.Polygon({
                  path: google.maps.geometry.encoding.decodePath(
                  response.isochrone3),
                  levels: decodedLevels,
                  strokeColor: "#FF0000",
                  strokeOpacity: 1.0,
                  fillColor: '#7F7F7F',
                  fillOpacity: 0.3,
                  strokeWeight: 2,
                  map: map
            });
      });
}
```

As one can see, we define three polygon regions based on the JSON data returned by the request, with different colors.

This is the final result in a browser:

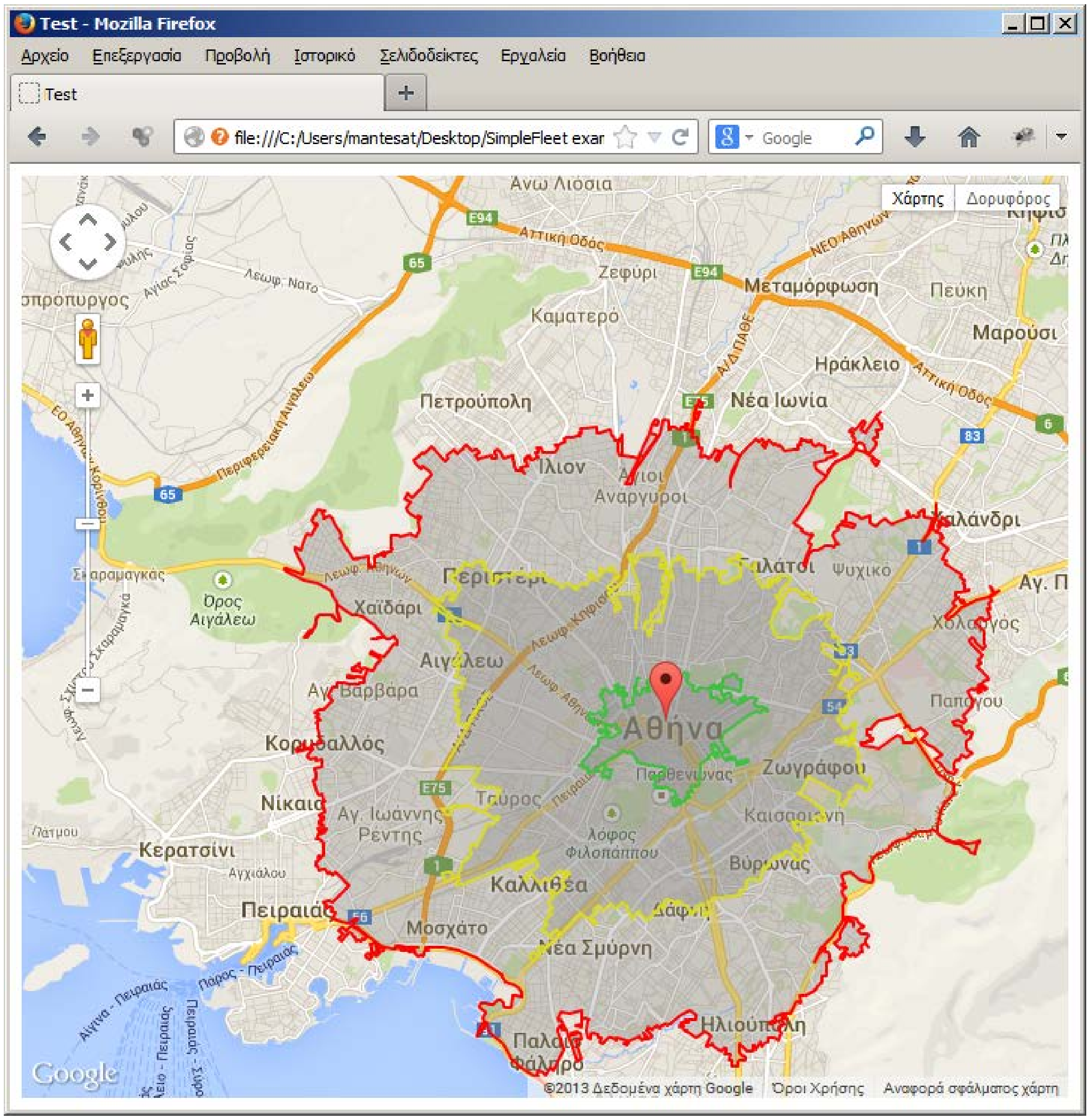


***Image 5**: Isochrones*

This is a very good example that we can easily add our traffic (or fleet) related data on a Google Maps website, which in fact is a very popular choice among web cartographers.

5 Conclusion and future work

In this deliverable an attempt was made to a) describe the new functionality added to the TrafficAPI and, most importantly, b) introduce a small set of sample code snippets or applications that could be useful to developers and users of the TrafficAPI services and the TrafficSDK client libraries. A lot of effort was put in implementing examples in as many programming languages or application environments as possible (although we left out some programming languages and frameworks that we thought are not too common, like C or Ruby). The major effort regarding the sample applications described earlier was to show that at any language or programming environment, for a developer to use the TrafficAPI services (by optionally using programming environment-related TrafficSDK client libraries), the main tasks are the following three:

1. Initiate an HTTP GET request using the proper REST URL and parameters to the TrafficAPI service

2. Use the TrafficSDK client libraries generated by Enunciate to parse the response into well defined entity objects (or just parse directly the JSON/XML response), and

3. Add some custom logic to it!

Regarding the first task, by now it has been clearly shown that at any programming language or application development framework, with or without the help of external utility libraries, performing an HTTP GET request is more or less an easy task (in all cases we saw that this was a built in feature). This confirms the design conclusion that by using REST web services we can serve all needs through one point and one server codebase.

Regarding the second task, we saw that in most cases, Enunciate-generated TrafficSDK client libraries make the job of bringing textual JSON or XML responses into objects that can be more easily handled, relatively easy. However, there were some examples that the TrafficSDK client libraries were not needed. In those cases, most environments could use already available JSON or XML parsing libraries to handle data directly.

Of course, the third task is the developer’s job. We tried to give some very simplistic code examples about how a fleet manager could add functionality without hassle. The rest depend on each case’s individual needs. Note that, soon most of these “Hello Fleet” apps will be uploaded to a site in order to be publicly available.

As for the future work on this field, the SimpleFleet DoW includes the design and implementation of the *iFleet* app, an application in IOS built for iPhone/iPad. The target is to incorporate all the functionality displayed to these small apps into iFleet and we are confident that we will be successful. Of course, work on the TrafficAPI will continue, by adding new features.

6 References

[Enunciate] A REST framework implementation. Online at: *http://enunciate.codehaus.org/*

[JAXB] Java Architecture from XML Binding Online at *http://www.oracle.com/technetwork/articles/javase/index-140168.html*

[JAX-RS] Online at: *http://jax-rs-spec.java.net/*

[Jersey] Java JAX-RS reference implementation. Online at: *http://jersey.java.net/*

[REST] A web service architecture. Online at: *https://en.wikipedia.org/wiki/Representational_state_transfer#RESTful_web_APIs*

[Spring] An application framework. Online at: *http://spring.io/*